Parallel Construction of Dialectical Trees for Defeasible Logic Programming[1]

(Preliminary Report)

Alejandro J. García Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Departamento de Ciencias de la Computación,
UNIVERSIDAD NACIONAL DEL SUR
Av.Alem 1253 – (8000) Bahía Blanca, ARGENTINA. FAX: (54) (291) 4595136
e-mail: ccgarcia@criba.edu.ar grs@criba.edu.ar

KEYWORDS: Defeasible Reasoning, Argumentation, Parallelism.

1 Introduction

Defeasible Logic Programming (DLP) [2] is an extension of logic programming based on the defeasible argumentation formalism developed in [12, 11, 6]. In DLP a query q succeeds when a *justification* for q is found. The defeasible argumentation formalism obtains justifications through a *dialectical analysis*, where arguments and counter-arguments are considered For implementing this dialectical analysis, a *dialectical tree* is used [6].

In [4], implicitly exploitable parallelism for DLP was studied. Since DLP is an extension of Logic Programming, the four kinds of parallelism already studied for traditional Logic Programming can be exploited [7]: OR-parallelism, independent and dependent AND-parallelism, and also unification parallelism. Besides, there are new sources of parallelism that can be implicitly exploited in a defeasible argumentation system: (1) several arguments for a conclusion q can be constructed in parallel, (2) once an argument $\mathcal{A}$ for q is found, defeaters for $\mathcal{A}$ can be searched in parallel, and (3) the dialectical analysis between arguments and defeaters, can also be done in parallel. In this work we propose an implementation for exploiting this last source of parallelism.

In order to develop an efficient implementation of DLP [2], an abstract machine called *Justification Abstract Machine* (JAM) has been designed as an extension of the Warren's abstract machine (WAM) [13, 8]. The JAM architecture has an instruction set, a memory structure and a set of registers expecially designed for building arguments, counterarguments and dialectical trees. The JAM inherits the WAM sequential architecture and the dialectical analysis is done sequentially, building the dialectical tree in a depth firsta approach.

In the last 15 years, several models for exploiting parallelism in Logic Programming has been developed [7], and several parallel extension for the WAM have been proposed. In [9], an efficient parallel execution model for logic programs was proposed, and the

[1]This work was partially supported by the Secretaría de Ciencia y Técnica, Universidad Nacional del Sur and Fundación OSDIC.

RAPWAM, an abstract machine for restricted and-parallel execution of logic programs was designed as an extension of the WAM (see also [10]).

The goal of this work is to introduce an implementation for constructing dialectical trees in parallel. We propose an implementation that extends the JAM, and is a combination between RAPWAM and JAM techniques. One of the main design objectives is to make these techniques compatible with those used in high performance sequential implementations.

This paper is organized as follows. First we introduce the main ideas of RAPWAM, then we describe a distribute model for building a dialectical tree in parallel. and finally we propose an extension of the JAM.

2 Restricted And-parallelism implementation

In Logic Programming, once a rule "$Head \leftarrow B_1, B_2, . . , B_n$" is selected during execution, *And-parallelism* consists on solving the body queries $B_1, B_2, \quad , B_n$ in parallel. However, given a rule as "$p(X, Z) \leftarrow q(X, Y), r(X, Z)$", there could be *variable bindings conflicts* if the queries q and r are executed in parallel and each execution binds the shared variable X with different terms. *Restricted And-Parallelism* (RAP) [1] is a technique which deals with these variable bindings conflicts by combining a compile-time analysis of the clauses involved, with simple checks performed on variables at run-time. Since our interest is focused only on the implementation of RAP as an abstract machine, we refer the interested reader to [1, 9, 10] for more details about RAP

Although logic programs can present considerable opportunities for And-Parallelism, there are always code segments requiring sequential execution. Therefore, parallel implementation techniques should extend those used in high performance sequential implementations. RAPWAM is an abstract machine for restricted and-parallel execution of logic programs, designed as an extension of WAM, so sequential execution is still as fast and space efficient as in WAM implementations (except for some minimal run-time checks).

The lack of space prevents us from fully describing the WAM and RAPWAM here. Instead we will only point out those basic concepts which are necessary for the understanding of our extensions. Therefore, we will descrive only the memory architecture of these abstract machines.

The WAM has four main memory areas:

1. the *Code* area, which holds the compiled program,
2. the *Heap*, which holds the terms produced by unification,
3. the *Stack*, which holds *environment registers* for supporting the execution chain, and *choice point registers* for implementing backtracking,
4. and the *Trail* where references to variables which need to be undone upon backtracking are stored.

In the WAM, choice points are created only when they are needed. Identifying the most recently choice point is immediate, since it is always pointed by a machine register B. The top of the Heap is stored in the choice point and updated upon backtracking.

Thus, the data just made obsolete by the failure that caused the backtracking is discarded. Prolog relies heavily on this retrieval of space during backtracking in order to avoid garbage collection.

In RAPWAM, each processor is equivalent to a standard WAM, except for

a) the inclusion of "*Parcall Frames*" in the local stack together with environments and choice-points, and

b) the addition of a "*Goal Stack*" to the memory areas.

Each processor has a Goal Stack where goals which are ready to be executed in parallel are pushed on to. Each entry in the Goal Stack is called a Goal Frame and contains all the necessary information for starting the remote execution of a goal. Suppose that the program rule "$Head \leftarrow Body$" is selected for execution, and that the queries $B_1, B_2, . \quad , B_k \in Body$ could be executed in parallel; then, instead of starting a sequential execution of B_1, and then B_2, and so on, the queries $B_1, B_2, . \quad , B_k$ are pushed on to the Goal Stack. Therefore, a goal can be "stolen" from the Goal Stack by any remote processor, which will copy the information of the Goal Frame, and will start execution from there. For avoiding idle waiting, a processor can also pick up a goal from its own Goal Stack.

Entries in the Goal Stack completely disappear after they are "picked up" by processors. However, the Parcall Frames are updated remotely and therefore they always have the information about the parallel activities of the children processors. Thus, the appropriate actions during backtracking can be selected.

3 Building a Dialectical Tree in Parallel

In Defeasible Logic Programming a query `h` succeeds if the supporting argument $\mathcal{A}$ for `h` is not defeated; $\mathcal{A}$ then becomes a *justification*. In order to verify whether an argument $\mathcal{A}$ is non-defeated, its associated counter-arguments $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_k$ are examined, each of them being a potential (defeasible) reason for rejecting $\mathcal{A}$. If any $\mathcal{B}_i$ is better than (or unrelated to) $\mathcal{A}$, then $\mathcal{B}_i$ is a candidate for defeating $\mathcal{A}$.

Since defeaters are arguments, there may exist defeaters for defeaters, and so on. In order to obtain a justification for a given query, a dialectical analysis is needed. The PROLOG program of Figure 1 shows the specification of this analysis (\+ stands for PROLOG's negation as failure).

```
justify(Q) :- find_argument(Q,A),
              \+ defeated(A)
defeated(A):- find_defeater(A,D),
              \+ defeated(D)
```

Figure 1: Justification specification

The specification of Figure 1 leads, in a natural way, to the use of trees to organize our dialectical analysis. In order to accept an argument $\mathcal{A}$ as a justification for q, a

tree structure can be generated. The root of the tree will correspond to the argument $\mathcal{A}$ and every inner node will represent a defeater (proper or blocking) of its father. The leaves in this tree correspond to non-defeated arguments. This structure is called a *dialectical tree.*

Definition 1 (Marking of a dialectical tree) *Let $\mathcal{A}$ be an argument for a literal h, and $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ be its associated dialectical tree. Nodes in $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are recursively marked as* defeated *or* undefeated *nodes (D-nodes and U-nodes respectively) as follows.*

1. *Leaves of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are* U-nodes.
2. *Let $\langle \mathcal{B}, q \rangle$ be an inner node of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$. Then $\langle \mathcal{B}, q \rangle$ will be an* U-node *iff every child of $\langle \mathcal{B}, q \rangle$ is a* D-node. *In contrast, the node $\langle \mathcal{B}, q \rangle$ will be a* D-node *iff it has at least an* U-node *as a child.*

Figure 2 shows the graphical representation of a marked dialectical tree for $\langle \mathcal{A}, h \rangle$, where every node (argument) is represented by a triangle.

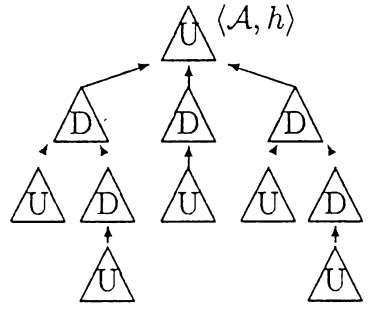

Figure 2: Marked Dialectical Tree

Definition 2 (Justification) *Let $\mathcal{A}$ be an argument for a literal h, and let $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ be its associated dialectical tree. The argument $\mathcal{A}$ will be a justification for a literal h if the root of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ is a U-node.*

In the current (sequential) system the dialectical tree is explored in depth-first order. However, given an argument $\mathcal{A}$, its defeaters can be computed in parallel. Therefore, the dialectical tree can be computed in parallel, if every time a node (argument) is obtained, its children (defeaters) are computed in parallel. This parallel process will give to the search tree a breadth-first flavor.

3.1 Parallel Execution Model for the Dialectical Tree

If the dialectical tree is computed in parallel, then the control of the justification process will be distributed over the nodes. Thus, the marking procedure of D-nodes and U-nodes will be done by message passing between the nodes of the dialectical tree. Next, we will introduce a model for building justifications in parallel. For the sake of clarity we will separate the construction of the dialectical tree, from the marking

procedure of the nodes. Our design goal is to delegate as much as possible work to the children process.

a) Parallel construction of the dialectical tree:
After constructing an argument $\mathcal{A}$ by a local processor P, the following actions will be carried out:

1. The set $Co(\mathcal{A})$ of potential counter-argument points will be obtained by the local processor P
2. For every $l \in Co(\mathcal{A})$ the construction of arguments for $\bar{l}$ will be called in parallel. These arguments are potential defeaters, and some of them could be the children of $\mathcal{A}$.
3. Once, the parallel calls are fired, the local processor P could also execute one of these parallel calls in order to avoid an idle waiting.
4. When a remote processor finish its execution, informs its father about its status: failure (no argument or D-node), or success (U-node) (see below).
5. Once at time all the children processors have finished, the local processor can evaluate the status of the argument $\mathcal{A}$ that has been built, and then informs this status to its father processor (see below).

The previous algorithm indicates how to build the dialectical tree. However, we also need to mark every node in order to know whether the root node is a U-node.

b) Parallel marking procedure of a dialectical tree.
In order to mark a node as U-node or D-node, the following criteria are used:

1. If a node $\mathcal{B}$ (argument) has no defeaters then $\mathcal{B}$ sends a message to its father indicating its success (it becomes a U-node).
2. If a node $\mathcal{A}$ receives from one of its children $\mathcal{B}$ the message that $\mathcal{B}$ is a U-node, then (as $\mathcal{B}$ defeats $\mathcal{A}$): (1) the node $\mathcal{A}$ becomes a D-node, (2) $\mathcal{A}$ sends a message to its father to inform that it is now a D-node, and (3) in order to prune the dialectical tree, $\mathcal{A}$ sends a message to its children that are still alive, to abort their dialectical process.
3. If every 'potential' defeater of $\mathcal{A}$ has failed, (no argument could be constructed or the argument was defeated) then $\mathcal{A}$ becomes a U-node, and sends a message to its father indicating this.

3.2 Implementation

As explained before, an efficient implementation of DLP [2], has been designed as an extension of the WAM: the *Justification Abstract Machine* or JAM for short. The JAM architecture has an instruction set, a memory structure and a set of registers especially designed for building arguments, counterarguments and the dialectical tree.

The JAM inherits the WAM sequential architecture and the dialectical analysis is done sequentially, building the dialectical tree in depth first. In this section we will show how the implementation techniques of the RAPWAM can be used for extending and modifing the JAM, in order to have a parallel justification procedure. We will call this extension P-JAM.

In P-JAM, each processor is equivalent to a standard JAM, except for

a) the inclusion of "*Defeaters Frames*" in the local stack together with environments and choice-points,

b) the addition of a "*Goal Stack*" to the memory areas,

c) and new instructions for dealing with parallel calls, and message passing.

In P-JAM, each processor has a Goal Stack where goals which are ready to be executed in parallel are pushed on to. These goals are actually the complement of the points of attack in the argument that the local processor has just constructed. Each entry in the Goal Stack is called a Goal Frame and contains all the necessary information for starting the remote execution of a goal that looks for a defeater. Following the parallel execution model for building the marked dialectical tree, every time a processor finish the construction of an argument $\mathcal{A}$, the P-JAM instruction **prepare_par-defeat** will do the following:

1. Looks in the TFT for N points of attack (each temporary fact).
2. For each point of attack "P" creates JAM code called "defeatK_at_pointP"
3. Creates the Defeaters Frame called D-Frame (like the Parcall Frame) with N slots
4. Loads the Goal Stack
5. and finally loads the last goal to be executed locally

Suppose that an argument $\mathcal{A}$ has the following points of attack: $p_1, p_2, \ldots, p_k$, then, instead of starting a sequential execution for looking for a defeater of p_1, and then p_2, and so on, the complements of the queries $p_1, p_2, \ldots, p_k$ are pushed on to the Goal Stack. Therefore, a goal (point of attack) can be "stolen" from the Goal Stack by any remote processor, which will copy the information of the Goal Frame, and will start the execution from there. As in the RAPWAM, a goal can be picked up from its own Goal Stack by the local processor avoiding idle waiting.

Since entries in the Goal Stack completely disappear after they are "picked up" by the remote processors, then the Defeaters Frames are updated remotely and therefore have the information of the parallel activities in the children processors. Thus, the appropriate actions for marking a node as D-node or U-node can be carried out.

Sometimes, a processor will finish its own work and will need to wait for the information of its children processors, while its Goal Stack is not empty. In this case, the instruction **pop_pending_defeaters** executed locally avoid idle waiting: if there are pending goals in the Goal Stack and there is not a successful defeater, then pop another goal from the goal stack and update D-frame properly, otherwise continue with next instruction.

If the Goal Stack of a processor is empty, but the processor has to wait for information from its children, then the instruction `wait_for_defeaters` could also avoid idle waiting popping another goal from some other goal stack. However, if at least the D-frame contains one successful defeater then the node of the local processor becomes a D-node and the pending defeaters could be killed.

Finally when the instruction `inform_father D` is executed, if all the defeaters slots in current D-Frame are fail the father D-frame at address "D" will be updated with "success" or otherwise they will be updated with "fail"

References

[1] Doug DeGroot. Restricted And-Parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pages 471–478, 1984.

[2] Alejandro J. García. *Defeasible Logic Programming: Definition and Implementation* (MSc Thesis). Departamento de Cs. de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, July 1997

[3] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming. Technical report, Computer Science Department, Universidad Nacional del Sur, October 1998. Technical Report GIIA-1998-20.

[4] Alejandro J. García and Guillermo R. Simari. Sources of parallelism in defeasible logic programming. In *Proceedings of the IV Congreso Argentino en Ciencias de la Computación*, October 1998.

[5] Alejandro J. García and Guillermo R. Simari. Strong and default negation in defeasible logic programming. In *In proceedings of the Fourth Dutch-German Workshop on Nonmonotonic Reasoning Techniques and Their Applications, DGNMR'99.* Institute for Logic, Language, and Computation, Amsterdam, The Netherlands, March 1999.

[6] Alejandro J. García, Guillermo R. Simari, and Carlos I. Chesñevar. An argumentative framework for reasoning with inconsistent and incomplete information. In *Workshop on Practical Reasoning and Rationality.* 13th biennial European Conference on Artificial Intelligence (ECAI-98), August 1998.

[7] Gopal Gupta, Khayri, A.M. Ali, Manuel Hermenegildo, and Mats Carlsson. Parallel execution of prolog programs: A survey. Technical report, Department of Computer Science, New Mexico State University, 1994. http://www.cs.nmsu.edu/lldap/pub_para/survey.html.

[8] Aït-Kaci Hassan. *Warren's abstract machine, a tutorial reconstruction.* MIT Press, 1991.

[9] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer

Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[10] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[11] Guillermo R. Simari, Carlos I. Chesñevar, and Alejandro J. García. The role of dialectics in defeasible argumentation. In *Anales de la XIV Conferencia Internacional de la Sociedad Chilena para Ciencias de la Computación.* Universidad de Concepción, Concepción (Chile), November 1994.

[12] Guillermo R. Simari and Ronald P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence*, 53:125–157, 1992.

[13] David Warren. An abstract prolog instruction set. Technical report, SRI International, Menlo Park, CA, October 1983. Technical Note 309.